MODIS
Software Development Standards and Guidelines

April 21, 1997

SDST-022

Revision C

MODIS Software Development Standards and Guidelines

Reviewed By:

_____

James Firestone, General Sciences Corporation/SAIC Date
SDST STIG Lead

_____

Jenny Glenn, General Sciences Corporation/SAIC Date
SDST Systems Analyst

_____

Barbara Putney, GSFC/Code 920 Date
MODIS Systems Engineer

_____

Catherine Harnden, GSFC/Code 920 Date
MODIS SDST Deputy Manager

_____

Laurie Schneider, General Sciences Corporation/SAIC Date
SDST R&QA Manager

_____

Tom Piper, General Sciences Corporation/SAIC Date
MODIS Task Leader

Approved By:

_____

Edward J. Masuoka, GSFC/Code 920 Date
MODIS SDST Manager

Change Record Page

This document is baselined and has been placed under Configuration Control. Any changes to this document will need the approval of the Configuration Control Board.

Document Title: MODIS Software Development Standards and Guidelines

Document Date: February 15, 1995

Issue	Date	Page Affected	Description
Original	2/15/1995	All	Baseline Review
Change 1	10/25/95	2 - 9, 11, 13 - 16	CCRs: 026, 049
Revision A	3/18/96	6 (para. 5.1.3)	CCR 125
Change 1	4/26/96	3, 7	CCRs: 131, 133
Change 2	7/13/96	17	CCRs: 128, 129, 152
Revision B	8/13/96	6,7, 8, 9, 11,12, 14, 15, 17	CCR 179
Change 1	11/18/96	14, 16	CCR 254
Revision C	4/21/97	All	CCRS: 287, 290

MODIS Software Development Standards and Guidelines

Table of Contents

MODIS Software Development Standards and Guidelines

1. INTRODUCTION

1.1 Purpose and Scope

This document defines and describes the standards and guidelines to be used for developing and maintaining software for use at the MODIS Team Leader Computing Facility (TLCF) and for delivery to the EOSDIS Project. Included are the tools and environment to be used at the SCFs, documentation standards, programming standards, and integration standards.

The standards and guidelines that are in this document are intended to facilitate the integration of the software into the EOSDIS Core System (ECS), to make the software more easily maintainable in the future, and to assist in the portability of the code. A standard must be followed. A guideline is a recommended practice.

1.2 Relationship to Project Standards

This document contains the EOSDIS document 423-16-01, Data Production Software and Science Computing Facility (SCF) Standards and Guidelines in the Attachment, which will be referred to as the Project Standards and Guidelines (PSG). This document is inclusive of or supersedes all the standards and most of the guidelines in the PSG. For easy reference, if there is a relationship between an item in this document and the PSG, it is noted in the text.

1.3 Waivers

When the project waiver policy is specified, we will adopt it for any item that does not meet the project standards. Requests for waivers to any of the standards should be sent to the Science Software Transfer Group (SSTG) member who is integrating the algorithm.

1.4 Document Status and Updates

This document has been updated to more closely follow Revision A of the PSG, dated October 1996. The major changes are:

1. Renumbered sections and standards, as appropriate.
2. Changed Appendix A to Attachment.

3. The following Standards (contained in Revision B) have been deleted:
   - Standard 5.1.1
   - Standard 5.1.2
   - Standard 5.3.1
   - Standard 7.1.2
   - Standard 7.2.5
   - Standard 7.3.1
   - Standard 7.4.1
   - Standard 7.4.2
4. The following Standards (contained in Revision B) have been changed to Guidelines:
   - Standard 5.1.4 is now Guideline 6.1.3
   - Standard 5.2.1 is now Guideline 6.2.3
   - Standard 5.2.2 is now Guideline 6.2.4
   - Standard 5.4.1 is now Guideline 6.5.1
   - Standard 5.6.1 is now Guideline 6.6.1
   - Standard 7.1.1 is now Guideline 8.1.4
   - Standard 7.2.4 is now Guideline 8.2.3
5. Created C Coding Standard regarding comparison of floating variables for equality (see Standard 5.1.1).
6. Revised Standard 5.3.2 regarding branching (see Standard 5.2.1).
7. Consolidated coding standards for FORTRAN 90 and FORTRAN 77 in Section 7.
8. Revised Standard 7.1.3 regarding comparison for strict equality (see Standard 7.1.1)
9. Consolidated coding guidelines for FORTRAN 90 and FORTRAN 77 in Section 8.
10. Deleted Section 9 FORTRAN 77 Coding Standards (see Section 7).
11. Deleted Section 10 FORTRAN 77 Coding Guidelines (see Section 8).
12. The Script Coding Standards and Guidelines are the same as those in the PSG (see Section 9).
13. The Integration Standards and Guidelines are contained in the MODIS Version 2 Science Computing Facility Software Delivery Guide (see Section 10).

2. SCF STANDARDS AND GUIDELINES

All SCF Standards and Guidelines are accepted as they are stated in Section 2 of the PSG. Refer to the Attachment, Section 2 of the PSG for these standards and guidelines.

All MODIS code will follow the ESDIS Standards and Guidelines (see the Data Products Software and SCF Standards and Guidelines, Revision A (October 1996) found in the Attachment of this document). Standards have not been repeated in this section of the document. Only standards above those dictated by ESDIS are in this section.

3. DOCUMENTATION STANDARDS

Documentation that is to be delivered from the SCF to the TLCF may be done in Microsoft Word, WordPerfect, or ASCII. Documentation from the TLCF to the Project will be done in Word or WordPerfect. ASCII will be used for packing lists and similar text files.

4. PROGRAMMING STANDARDS AND GUIDELINES

This section begins with standards and guidelines for commenting code. Subsequent sections contain standards and guidelines for C and FORTRAN coding. No MODIS code will be done in Ada, therefore, no standards or guidelines were included for that language.

4.1 Commenting Standards

The commenting standards consist of the standard module prolog and a set of guidelines for documenting the rest of the code.

4.1.1 Module Identification Standard

4.1.1.1 Source Code

To allow identification of individual items of code, a header (prolog), as defined below, shall be inserted at the top of each module in the production software. A module is a main program, subroutine, procedure, function, etc. This header should also be included at the top of insert/include files, with the exception that the blocks relating to input and output parameters are omitted. Prologs are required by the MODIS coding standards to facilitate the integration of the science code. The SSTG will rely heavily on information in the prologs to direct them in the integration of the SDP Toolkit.

NOTE: The example uses the C language style of comments and would need to be converted for other languages. An exclamation mark "!" is used in the following example to indicate all sections of the prolog that are mandatory, and the line numbers are to be used in conjunction with the descriptions that follow. Line numbers are not required in the actual code. However, the "!" is mandatory so that automated tools used in the DAAC can extract portions of the prolog.

For heritage code, where the generation of this header information for every module is an unreasonable burden, an alternative approach based on compilation units can be used. A single header can be used to record the description and version history for all modules contained within a file which is compiled as a unit. In this case the only requirement for each module is to describe each input/output parameter (not globals). Although this alternative approach is acceptable for heritage code, the former approach, with a header for each module, is required for all new code.

The inclusion of the headers will allow automated tools within the DAAC I&T environment to manipulate the software items and will ease understanding by the I&T team.

Templates for module headers and include file headers can be accessed through the PGS Toolkit interface software. The following is an example of a module header:

```
line no.
00   geonav(float pos[3],float smat[3][3],float coef[6],float sun[3],
01   int nsta,int ninc,int npix,float xlat[409],float xlon[409],
02   float solz[409],float sola[409],float senz[409],float sena[409])
03/*
04!C*********************************************************
05
06 !Description:    This subroutine calculates the sensor
07  orientation from the orbit position vector and input values of
08  attitude offset angles.  The calculations assume that the angles
09  represent the yaw, roll and pitch offsets between the local
10  vertical reference frame (at the spacecraft position) and the
11  sensor frame. The outputs are the matrix which represents
12  the transformation from the geocentric rotating to sensor
13  frame, and the coefficients which represent the Earth scan
14  track in the sensor frame. The reference ellipsoid uses an
15  equatorial radius of 6378.137 km and a flattening factor of
16  1/298.257 (WGS 1984).05
17
18 !Input Parameters:
19  pos[3]         satellite position
20  smat[3][3]   sensor orientation matrix
21  coef[6]        scan line coefficients
22  sun[3]         unit sun vector in geocentric rotating coordinates
23  nsta            number of first pixel to start with
24  ninc           increment between pixels for computations
25  npix           number of pixels in scan line
26
27 !Output Parameters:
28  xlat[409]      latitude values
29  xlon[409]     longitude values
30  solz[409]      solar zenith values
```

```
31  sola[409]      solar azimuth values
32  senz[409]     sensor zenith values
33  sena[409]      sensor azimuth values
34
35 !Revision History:
36  $LOG: geonav.c,v $
37  Revision 01.01  1993/10/12 10:12:28
38  Z. GREEN (zgreen@harp.gsfc.nasa.gov)
39  Initial delivery of software.  Modified to comply with ESDIS
40  standards. It was a breeze.
41
42 Revision 01.00  1993/04/29 17:12:28
43 A. SMITH (asmith@harp.gsfc.nasa.gov)
44 Initial debugged version, based on original FORTRAN 77
45 subroutine developed in 1983 by C. ADAMS of the TELLUS
46 project.
47
48 !Team-unique Header:
49  References and Credits
49a Design Notes
50 !END************************************************
     */
            <code follows here>
```

The following notes describe how to use the header block, using the above example as a reference.

Line 00: Name of main procedure, include file or subroutine/ procedure/ function. If this is not the main procedure or an include file, it should contain the function statement.

Line 04: Start of prolog. Initial marker can take the following values:

!FXY - contains FORTRANXY (XY = 77 or 90) executable statements

!C - contains C executable statements

!FXY-INC - FORTRANXY (XY = 77 or 90) include file

!C-INC - C include file

Line 06: A concise but complete summary of the overall function of the module. Any references for methods and/or algorithms should be included. Use as many lines as necessary.

Line 18: Header for input parameters

Line 19-25: Input parameters (not global variables) in the order they are presented to the module with a short 1-2 line description of the parameter (and its units where appropriate). Global variables should be described in the module in which they are declared (i.e., only once).

Line 27: Header for output parameters.

Line 28-33: Output parameters (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters.

Line 35: Start of Revision History Log.

Line 36: If you are using an automated tool for revision control, you should insert any statements required immediately after the Modification History Log Header.

Line 37-46: Each revision should contain as a minimum the revision number, date, time, person, and email address, with a short description of ALL the changes made. The first revision should include the original author of the code. Revisions should be ordered with the latest first. [Note: this revision information can be supplemented with more detailed comments in the code referencing the revision number].

Use a revision numbering scheme where the revision number is in the format "nn.mm" where "mm" is updated each time a change is made to the module and "nn" is updated when the function of the module changes or the algorithm/method is changed. [Note: The release number for the data production software is not related to the revision numbers on individual modules - the release number scheme should be determined by the development team. The date associated with a release is more meaningful than a release number to those outside the development team.]

Line 48: Start of Team-unique Header. SDST will define a standard MODIS Team header that will appear in all prologs for code. This is required by the ESDIS for the software released to the DAAC.

Line 49: Any references and credits should be noted here.

Line 49a: Any design notes, limitations, etc. should be noted here.

Line 50: End of source code prolog.

4.1.1.2 Script file

A prolog similar to the one for source code will be used.

4.1.2 Commenting Guidelines

All code should have enough comments to make it understandable to another programmer. The following are some guidelines to follow as you comment the code.

4.1.2.1 Comments should be maintained to ensure their correctness.

4.1.2.2 Comments should account for a significant percentage of the lines in the source code.

4.1.2.3 Comments should appear before every major control construct (loops, if-thens, switches, etc.).

4.1.2.4 Above-the-line comments should be immediately above the line of code without separating; white space should be indented to the same column as the line of code.

4.1.2.5 Right margin comments should use enough white space to clearly separate comments from code and should all begin in the same column if they are commenting a block of statements.

4.1.2.6 Comment delimiting should be done in a consistent manner.

5. C CODING STANDARDS

The mandatory coding standards for code generated in the C language follow. (See Section 3 of the ESDIS Standards and Guidelines in the Attachment of this document.)

5.1 General Standards

5.1.1 Float and double variables shall not be compared for strict equality with other float or double variables (i.e., using == or !=). Float and double variables may be compared for strict equality with floating literals (e.g., a == 0.0), however, this practice is discouraged and should be used sparingly. (Supersedes PSG Guideline 3.3.13.) The rationale for this standard is portability.

5.1.2 Do not use pragma preprocessor directives. Different compilers may produce different implementations of these directives, or (if the directive is not recognized) no implementation at all. The rationale for this standard is portability.

5.2 Structuring/Style

5.2.1 Unconditional branching (GOTO) shall only reference a label further down in the code. (Supersedes PSG Guideline 3.3.7.) The rationale for this standard is maintainability.

5.3 Bit Manipulation

5.3.1 Structures shall not contain bit field elements. Compilers differ in whether bits are counted from left to right, or from right to left. The rationale for this standard is portability.

5.4 Functions

5.4.1 All user-defined functions shall be typed and prototyped. Functions which do not return a value shall be typed as void. (Supersedes PSG Guidelines 3.3.14 and 3.3.15.) The rationale for this standard is portability. An example of prototyping follows:

```
int GreatestCommonDenominator(int large_term; int small_term);
```

6. C CODING GUIDELINES

The following are suggested guidelines for C code. The book C Programming Guidelines by Thomas Plum is recommended as a resource for additional guidelines for writing portable, maintainable, and effective code.

6.1 General Guidelines

6.1.1 Free resources that have been allocated.

6.1.2 Use the sizeof operator to determine the length of a structure.

6.1.3 External names may be used with a size greater than the limit in ANSI of 6 characters, but not to exceed 31 characters. (Supersedes PSG Guideline 3.3.19)

6.2 Declarations

6.2.1 Declarations should be ordered consistently throughout the code. Use whatever method makes the most sense for the particular application. Pointers should immediately follow what they are pointing to. For example you could order them as follows:

1. . declaration of external variables in type order,
2. . declaration of local variables in type order,
3. . declaration of functions used in type order.

The type order could be char, short, unsigned short, long, unsigned long, float, double, enum, struct, union, void. (Supersedes PSG Guideline 3.3.4)

6.2.2 All pointer variables should be named in a consistent fashion. For example all pointer variables may consist of the name of the variable with an "_p" at the end (e.g., precip_p may be a pointer to precip).

6.2.3 Shared global definitions should be put in a header file. The rationale for this guideline is maintainability.

6.2.4 System library functions should not be declared; the appropriate header files should be included.

6.3 Structuring/Style

6.3.1 There should be sufficient white space to make the code readable. Blank lines between each major code structure are encouraged.

6.3.2 Use a consistent style to highlight code structure and increase readability. This includes a consistent and descriptive naming convention for variables. All structures will use a consistent indentation scheme within a module for each succeeding level of the structures.

6.3.3 Names of symbolic constants and user-defined macros in #define statements should be in upper-case. All other names should not be in all upper-case. Use of mixed case is encouraged.

6.3.4 Every header should prevent multiple inclusions of itself. For example:

```
/*   This is a dummy header file with the name dummy.h that shows how to
avoid multiple declarations of variables and functions.
Note that the name of the header file is used as a parameter that is
defined or undefined depending on whether it has been invoked  in  the
module processing.
The use of underlines and captial letters make it unlikely  that  this
parameter will be confused with others in the program.
*/
#ifndef __DUMMY_H_      /* see if this header has been invoked before */
#define __DUMMY_H_      /* if it has not, then define it's "definition
variable" so it will not be invoked again by the including module.  Then
proceed to process the header.*/
/*
     header code goes here, including necessary #include
*/
#endif
```

6.4 Operators

6.4.1 Use explicit type casts. The rationale for this standard is portability. (Supersedes PSG Guideline 3.3.12.) For example:

```
long integer_variable;
long * integer_pointer;
double floating_point_var;
integer_pointer = &integer_variable; /* per guideline */
integer_pointer = &floating_point_var; /* contrary to guideline */
integer_variable = (long)floating_point_var;/* per guideline */
```

6.5 Bit Manipulation

6.5.1 Bitwise operations should not be used on signed numbers. The rationale for this standard is portability.

6.6 Error Checking

6.6.1 All return call statuses should be checked and the appropriate actions taken. The rationale for this guideline is maintainability.

7. FORTRAN CODING STANDARDS

The mandatory coding standards for code generated in either FORTRAN 77 or FORTRAN 90 language follow. (See Section 4 of the PSG in the Attachment.)

NOTE: Standards applying only to FORTRAN 77 or FORTRAN 90 are so marked in the text.

7.1 General Standards

7.1.1 Double precision and real variables should not be compared for strict equality with other double precision or real variables (i.e., using .EQ., .NE., ==, /=). Double precision and real variables may be compared for strict equality with double precision or real literals (e.g., a .EQ. 0.0), however, this practice is discourages and should be used sparingly. (Supersedes PSG Guideline 4.3.13.) The rationale for this standard is portability.

7.2 Declarations

7.2.1 All variables shall be explicitly declared. The IMPLICIT NONE statement shall appear in each routine. The rationale for this standard is maintainability. This standard applies to FORTRAN 90 code only; however, it should be applied to FORTRAN 77 code as a guideline.

7.2.2 All variables shall be initialized prior to use. (Supersedes PSG Guideline 4.3.3) The rationale for this standard is portability.

7.2.3 PARAMETER variables shall not be redefined. (Supersedes PSG Guideline 4.3.5) The rationale for this standard is maintainability.

8. FORTRAN CODING GUIDELINES

The following are suggested guidelines for FORTRAN 77 and FORTRAN 90 code. At the current time we have no book to recommend for additional guidelines for writing portable, maintainable, and effective code.

8.1 General Guidelines

8.1.1 Format statements should be collected together, preferably at the end of the routine.

8.1.2 Only SAVEd variables are guaranteed to retain their value between module calls; all other variables should be assumed to be undefined for each access of a module. Except for loop indices, variables should not be re-used within a routine.

8.1.3 Free all resources that have been allocated.

8.1.4 Compiler options should not be used to perform functions that can be handled by the language. For example, SAVE statements should be used to statically allocate local variables rather than a compiler option such as -static on the SGI FORTRAN compiler. The rationale for this guideline is portability.

8.2 Declarations

8.2.1 All pointer variables should be named in a consistent fashion. For example all pointer variables may consist of the name of the variable with an "_p" at the end (e.g., precip_p may be a pointer to precip).

8.2.2 A pointer should not point to variables of types different from that indicated by the pointer declaration.

8.2.3 COMMON Blocks should be named. The rationale for this guidline is readability.

8.2.4 EQUIVALENCE statements should not be used.

8.3 Structuring/Style

8.3.1 There should be sufficient white space to make the code readable. Blank lines between each major code structure are encouraged.

8.3.2 Use mixed case rather than all upper case.

8.3.3 Computed and arithmetic GOTOs may be used but are strongly discouraged. (Supersedes PSG 4.3.10)

8.4 Labeling

8.4.1 A consistent labeling scheme will be used, with labels increasing in value through a module. (Supersedes PSG Guideline 4.3.12) The rationale for this standard is maintainability.

9. SCRIPT CODING STANDARDS

The script coding standards and guidelines are contained in Section 7 of the PSG (see Attachment).

10. INTEGRATION STANDARDS

Integration standards and guidelines are contained in the MODIS Version 2 Science Computing Facility Software Delivery Guide.

ATTACHMENT: EOSDIS DOCUMENT 423-16-01, REVISION A

This Attachment contains the Data Production Software and Science Computing Facility (SCF) Standards and Guidelines, EOSDIS document 423-16-01. The original document can be found on URL: spsosun.gsfc.nasa.gov/ESDIS_Docs.html.

423-16-01

Data Production Software and Science Computing Facility (SCF) Standards and Guidelines

Revision A

October 1996

[Image]

423-16-01

Data Production Software and Science Computing Facility (SCF) Standards and Guidelines

Approved by

_____	_____
Stan Scott ESDIS Science Software Manager GSFC - Code 505	Date
_____	_____
Steve Kempler Data Processing Manager GSFC - Code 505	Date
_____	_____
Dale Harris Associate Director for ESDIS Project GSFC - Code 505	Date

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

423-16-01

CHANGE RECORD PAGE

Issue	Date	Pages Affected	Description
Original	01/14/94	All	Baseline
CH01	02/15/95	i, ii, iii, iv, 1, 5, 6, 7, 8, 9, 10, 11, 14, 16, 17, 18	CCR 505-01-16-001
Rev A	10/15/96	All	CCR 505-01-16-002-A

423-16-01

LIST OF AFFECTED PAGES

Page No.	Revision	Page No.	Revision
Title	Rev A	4-6	Rev A
i	Rev A	5-1	Rev A
ii	Rev A	5-2	Rev A
iii	Rev A	6-1	Rev A
iv	Rev A	6-2	Rev A
v	Rev A	6-3	Rev A
vi	Rev A	6-4	Rev A
vii	Rev A	6-5	Rev A
viii	Rev A	6-6	Rev A
ix	Rev A	7-1	Rev A
x	Rev A	7-2	Rev A
1-1	Rev A	A-1	Rev A
1-2	Rev A	A-2	Rev A
2-1	Rev A	A-3	Rev A
2-2	Rev A	A-4	Rev A
3-1	Rev A	A-5	Rev A
3-2	Rev A	A-6	Rev A
3-3	Rev A	A-7	Rev A
3-4	Rev A	A-8	Rev A
4-1	Rev A	B-1	Rev A
4-2	Rev A	B-2	Rev A
4-3	Rev A	B-3	Rev A
4-4	Rev A	B-4	Rev A
4-5	Rev A	AB-1	Rev A
		AB-2	Rev A

CONTENTS

423-16-01

Data Production Software and SCF Standards and Guidelines

1. INTRODUCTION

1.1 PURPOSE

The purpose of these standards is, fundamentally, to avoid excess costs over the life cycle of the Data Production Software. The standards contained in this document are motivated by a need for maintainable and portable software.

1.2 SCOPE

The standards promulgated in this document apply to networked Science Computing Facilities (SCFs) and data production software that will be delivered for integration into the Data Processing Subsystem of the Earth Observing System (EOS) Data and Information System (EOSDIS) Core System (ECS) at the Distributed Active Archive Centers (DAACs). The scope of these standards explicitly excludes prototype code or supporting software that may be used in building Data Production Software but that will not be delivered to the DAAC.

The Project's policy for Earth Science Data and Information System (ESDIS) Science Team prohibited functions is presented in Appendix A.

The standards in this document are mandatory. Standards statements always include the word "shall". Guidelines found in this document are not mandatory, but are included as recommendations. The guidelines always include the word "should" or "may" in the statement.

1.3 AUTHORITY

This document is issued under the authority of the ESDIS Project (the Project). The original standards were developed by the ESDIS Project with the assistance and consensus of the Data Processing Focus Team (DPFT). The DPFT membership included several representatives from EOS Data Production Software Developers and DAACs.

1.4 WAIVERS

The Project may issue waivers to these standards for performance or heritage software on a case by case basis.

The Project's policy for ESDIS Science Team standards and waivers is presented in Appendix B.

1.5 EXTENSIONS OF STANDARDS

The Project anticipates that these standards may be extended in the future.

1.6 OFF-THE SHELF, THIRD-PARTY SOFTWARE POLICY

Existing off-the-shelf software from a third party (e.g., the McIdas library, MODTRAN,) are exempt from the module identification standard (Section 6). The science team is responsible for ensuring correct compilation and execution of the off-the-shelf software at the DAAC.

1.7 RELATED DOCUMENTATION

The following documents are referenced or contain policies or other matter that are binding upon the contents of this document:

a.) Functional and Performance Requirements Specification for the Earth Observing System Data and Information System (EOSDIS) Core System, GSFC 423-41-02, 16 February 1993

b.) Interface Control Document between EOSDIS Core System (ECS) and Science Computing Facilities (SCF), GSFC 505-41-33

c.) SDP Toolkit Users Guide, GSFC 505-16-03, or superseding versions

d.) Military Standard; Ada Programming Language, MIL-STD-1815A, 22 January 1983

e.) Military Standard; FORTRAN, MIL-STD-1753, 9 November 1978

f.) IEEE Standard for Information Technology, Portable Operating System Interface (POSIX), Part 2: Shell and Utilities, IEEE Std 1003.2-1992

g.) Programming Languages - C (revision and redesignation of ANSI X3.159-1989), ANSI/ISO 9899-1990

h.) American National Standard Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980 (E)

i.) American National Standard for Programming Language- Fortran- Extended, ANSI X3.198-1992.

2. SCF STANDARDS AND GUIDELINES

2.1 SCF HARDWARE

2.1.1 Intent

The intent of the SCF hardware standards is to control the number of versions of the Science Data Production (SDP) Toolkit that the Project must support for the Data Production Software teams.

2.1.2 Standards

2.1.2.1 SCF hardware shall be capable of running the software included in the SCF Software Standards.

2.1.3 Guidelines

2.1.3.1 SCF hardware hosting the SDP Toolkit should be compatible with hardware from the set of SDP Toolkit hosts established by the ECS Contractor. The rational for this guideline is to facilitate portability.

2.1.3.2 SCF hardware should be from the same vendor family as the target DAAC data processing computer. The rational for this guideline is to facilitate portability.

2.2 SCF COMMUNICATIONS (DELETED)

2.3 SCF SOFTWARE

2.3.1 Intent

The intent of this standard is to ensure correct operation of the SCF- DAAC interface.

2.3.2 Standards

2.3.2.1 SCFs shall have software that supports applicable network protocols and mechanisms described in Section 4 of the ECS-SCF ICD.

2.3.2.2 SCFs that are used to order data from the DAAC shall have software that provides the capability to use an ECS client.

2.4 SCF SECURITY

2.4.1 Intent

The intent of this guideline is to help establish minimum security requirements for networked SCFs. More elaborate security may be imposed locally.

2.4.2 Guidelines

2.4.2.1 Multi-user SCFs should use system security with procedures to establish and maintain, as a minimum:

a.) Passwords,

b.) User Accounts,

c.) User Permissions.

2.4.2.2 All passwords on networked, multi-user SCFs should adhere to the following:

a.) Passwords should consist of 6 or more characters, including at least one numeric or "special" character (such as a space or asterisk);

b.) Passwords should not be the user's login name, or circular shift of that name.

2.4.2.3 A password aging scheme should be employed, to force users to change passwords periodically (e.g., every 90 days).

2.4.2.4 The System Administrator of networked SCFs should:

a.) Reset all factory-set passwords immediately;

b.) Disable the Guest account;

c.) Either omit the Anonymous FTP account or restrict it to a specific directory;

d.) Assure the initial password for a new account is not a forename-surname or other usage of the user name;

e.) Minimize distribution of the root password.

2.4.2.5 The System Administrator of networked, multi-user SCFs should:

a.) Establish groups for users with read and execute permissions for group members;

b.) Remove global read/write access to any files which are not for public view;

c.) Remove global execute access to any directory which is not for public access.

2.4.2.6 The command ".netrc" should not be used (It would allow login without a password).

3. DATA PRODUCTION SOFTWARE STANDARDS AND GUIDELINES FOR THE C LANGUAGE

3.1 INTENT

The intent of this standard is to facilitate portability of data production software and to control costs at the DAACs.

3.2 STANDARDS

The mandatory coding standards for data production software generated in the C language follow.

3.2.1 Comply with ANSI Standard

The software shall comply with the ANSI standard specification for C (ANSI ISO 9899-1990). Vendor specific extensions to the standard shall not be used. The rationale for this standard is portability.

3.2.2 Use SDP Toolkit Calls

External calls from the operational data production software in the DAAC, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through SDP Toolkit calls. The software shall comply with the restrictions found in Appendix A. Data production software delivered to the DAAC shall be fully interoperable with the SDP Toolkit installation at the DAAC. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating the development effort.

3.3 GUIDELINES

3.3.1 ANSI Checking

It is strongly recommended that code be compiled by the IT at the SCF with the ANSI checking option on. The rationale for this guideline is portability.

3.3.2 File Inclusion

The <> and "" notation should be used for including standard C header files and programmer created header files, respectively. The rationale for this guideline is maintainability.

3.3.3 Initialize Variables

All variables should be initialized prior to use (except for static variables), i.e. no assumptions should be made that variables are initialized to zero. The rationale for this guideline is portability.

3.3.4 Order Declarations

Declarations should be ordered consistently throughout the code. An example of a consistent order is:

a.) declaration of module arguments in the same order as the argument list (before the opening brace of the function),

b.) declaration of external variables,

c.) declaration of local variables,

d.) declaration of functions used.

The rationale for this guideline is maintainability.

3.3.5 Naming Convention

A consistent and descriptive naming convention should be adopted. The rationale for this guideline is maintainability.

3.3.6 Integer Loop Control Variables

Loop control variables should be of INTEGER type. The rationale for this guideline is portability. Note: This guideline includes the subtypes of integer, such as char, short, long, etc.

3.3.7 Avoid GOTOs

Unconditional branching (GOTO) should only be used within nested structures and should only reference a label further down in the code. The rationale for this guideline is maintainability.

3.3.8 Consistent Style

Use a consistent style to highlight code structure and increase readability. The rationale for this guideline is maintainability.

3.3.9 Pointer Type

A pointer should have the same type as the variable it points to. The rationale for this guideline is portability.

3.3.10 Variable Value Retention

Static variables are guaranteed to retain their value between module calls; all other variables in a module should be assumed to be undefined for each access of a module. External variables also retain their values between module calls. The rationale for this guideline is portability.

3.3.11 Contiguous Use Assumption

Contiguous use of memory by arrays should not be assumed. The rationale for this guideline is portability.

3.3.12 Implicit Type

Avoid implicit type casts. The rationale for this guideline is portability. Note: C language processors will generate code to cast between the types on the left and right side of an assignment operator, but reliance on this implicit type conversion is not a good idea. The programmer should include explicit type casts:

```
long integer_variable;

long * integer_pointer;

double floating_point_variable;

integer_pointer = &integer_variable; /* per guideline */

integer_pointer = &floating_point_variable; /* contrary to guideline */

integer_variable = (long)floating_point_variable; /* per guideline */
```

3.3.13 Equality Comparisons

Float and double variables should not be compared for strict equality (i.e., using == or !=). The rationale for this guideline is portability.

3.3.14 Function Prototype

All functions should be typed (and preferably prototyped). The rationale for this guideline is maintainability. Note: the use of function prototypes significantly reduces interface errors in C. A function prototype tells the compiler the type of the function and the number and types of the arguments. An example follows:

```
int GreatestCommonDenominator(int large_term; int small_term);
```

3.3.15 Void Functions

Functions which do not return a value should be typed as void. The rationale for this guideline is maintainability.

3.3.16 Use "div" and "ldiv"

The div and ldiv functions in the C Standard Library should be used to obtain consistent values of remainders when the quotient is negative. The rationale for this guideline is portability, since the C language definition does not specify how to handle this situation, and at least two answers are possible. Note: in general static analysis cannot ensure that the quotient will never be negative so the library functions should be used instead of the % operator.

3.3.17 Rounding

The correct functioning of code should not depend on the rounding behavior of converting a long double to other floating types or double to a float. The rationale for this guideline is portability.

3.3.18 Exceeding ANSI C 32K Limit

Data objects (arrays or structures), of a size greater than the limit in ANSI C of 32K for data objects, may be used.

3.3.19 Exceeding ANSI C 6-Character Name Limit

External names, of a size greater than the limit in ANSI C of 6 characters, may be used.

3.3.20 Comply with POSIX Standard

The source code should comply with IEEE Standard 1003.1, POSIX-Part 1: System Application Program Interface (API) [C Language]. The rationale for this guideline is portability.

4. DATA PRODUCTION SOFTWARE STANDARDS AND GUIDELINES FOR THE FORTRAN LANGUAGE

4.1 INTENT

The intent of this standard is to facilitate portability of data production software and to control costs at the DAACs.

4.2 STANDARDS

The mandatory coding standards for data production software generated in the FORTRAN language follow.

4.2.1 FORTRAN Compiler

The FORTRAN compiler standard consists of the following parts:

a.) Science data production software shall comply with the ANSI standard specification for FORTRAN 77 (ANSI/X3.9-1978) or for Fortran 90 (ANSI X3.198-1992).

b.) Heritage FORTRAN 66 code shall be made to compile using FORTRAN 77 or Fortran 90 prior to delivery to the DAAC. The rationale for this standard is portability.

4.2.2 Use SDP Toolkit Calls

External calls from the operational data production software in the DAAC, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through SDP Toolkit calls. Data production software delivered to the DAAC shall be fully interoperable with the SDP Toolkit installation at the DAAC. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating the development effort.

4.2.3 FORTRAN 77, Fortran 90, and SDP Toolkit Calls

The FORTRAN 77 software shall comply with the restrictions found in Appendix A.

FORTRAN 77 code shall use only those SDP Toolkit calls that provide capabilities found in the FORTRAN 77 language. The rationale for this standard is that there will be no FORTRAN 77 bindings for SDP Toolkit capabilities requiring unique Fortran 90 features.

4.3 GUIDELINES

The first guideline is the extensions to the FORTRAN 77 language that follow. They were permitted prior to the development of the ESDIS waiver policy and are considered as an EOS community waiver.

4.3.1 FORTRAN 77 Extensions

The following extensions to FORTRAN 77 may be used:

a.) INCLUDE statement (MIL-STD-1753, Section 2.3),

b.) BYTE data type,

c.) DO WHILE (MIL-STD-1753, Section 2.2),

d.) ENDDO (MIL-STD-1753, Section 2.1),

e.) STRUCTURE data types,

f.) names up to 31 characters in length,

g.) IMPLICIT NONE statement (MIL-STD-1753, Section 2.4),

h.) block IF with ELSE IF and END IF,

i.) in-line comments,

j.) extended character set to include lower case letters, underscore ("_"), left and right angle bracket ("[" and "]"), quotation mark ("), exclamation point ("!"), percent sign ("%"), and ampersand ("&"),

k.) initialization of data in declaration,

l.) long line extensions beyond 72 characters per line.

m.) bit manipulation (MIL-STD-1753, Section 2.6), including: Inclusive OR, Logical AND, Logical Complement, Exclusive OR, Logical Shift, Circular Shift, Bit Extraction, Bit Move, Bit Testing, Set Bit, Clear Bit, and Bit Constants,

n.) EXIT ().

The ESDIS Project will attempt to procure FORTRAN 77 compilers for the DAACs that permit the above extensions to the ANSI standard language, but the DAAC compiler acceptance of these extensions can not be guaranteed. Those science teams possessing heritage code that uses most of these extensions may wish to consider converting the code to use a Fortran 90 compiler. The FORTRAN 77 language and several extensions are a subset of the Fortran 90 language.

4.3.2 Compatibility with the next FORTRAN Standard

Any constructs and features of the Fortran 90 language that are marked for removal in the next release of the FORTRAN standard should not be used (ANSI X3.198-1992, Section 1.6 & Annex B):

a.) Arithmetic IF.

b.) Real and double precision DO control variables and DO loop control expressions.

c.) Shared DO termination and termination on a statement other than END DO or CONTINUE.

d.) Branching to an END IF statement from outside its IF block.

e.) Alternate Return.

f.) PAUSE statement.

g.) ASSIGN and assigned GO TO statements.

h.) Assigned FORMAT specifiers.

e.) cH edit descriptor.

4.3.3 Initialize Variables

All variables should be initialized prior to use. The rationale for this guideline is portability.

4.3.4 Order Declarations

Declarations should be ordered consistently throughout the code. An example of consistent ordering is:

a.) declaration of module arguments in the same order as the argument list,

b.) declaration of global variables in COMMON (using INCLUDE files),

c.) declaration of local PARAMETERs (types and values),

d.) declaration of local variable types,

e.) declaration of user defined function types used,

f.) EXTERNAL declarations,

g.) INTRINSIC declarations,

h.) declaration of DATA values.

The rationale for this guideline is maintainability.

4.3.5 PARAMETER Variables

PARAMETER variables should not be redefined. The rationale for this guideline is maintainability.

4.3.6 COMMON Blocks

COMMON blocks should be inserted into the code using INCLUDE. The rationale for this guideline is maintainability.

4.3.7 Naming Convention

A consistent and descriptive naming convention should be adopted. The rationale for this guideline is maintainability.

4.3.8 Integer Loop Control Variables

Loop control variables should be of INTEGER type. The rationale for this guideline is maintainability.

4.3.9 Avoid GOTOs

Unconditional branching (GOTO) should only be used within nested structures. The rationale for this guideline is maintainability.

4.3.10 Avoid Computed and Arithmetic GOTOs

Computed and arithmetic GOTOs should not be used. The rationale for this guideline is maintainability.

4.3.11 Terminate DO-loops

DO-loops should be terminated with CONTINUE or ENDDO. The index of a DO loop should always be of integer type, and the index should not be modified inside the loop. The rationale for this guideline is maintainability.

4.3.12 Consistent Style

Use a consistent style to highlight code structure and increase readability. The rationale for this guideline is maintainability.

4.3.13 Equality Comparisons

Real and complex variables should not be compared for strict equality (i.e., using .EQ. or .NE.). The rationale for this guideline is portability.

4.3.14 Consistent Labeling

A consistent labeling scheme should be used. The rationale for this guideline is maintainability.

4.3.15 Generic Intrinsic Functions

Generic intrinsic functions should be used rather than type specific functions. The rationale for this guideline is portability.

4.3.16 Rounding

The correct functioning of code should not depend on the rounding behavior of converting a DOUBLE PRECISION or COMPLEX to other floating types. The rationale for this guideline is portability.

4.3.17 Comply with POSIX Standard

The FORTRAN 77 source code should comply with IEEE Standard 1003.9, POSIX FORTRAN 77, Language Interfaces, Part 1: Binding for System Application Program Interface (API). The rationale for this guideline is portability.

5. DATA PRODUCTION SOFTWARE STANDARDS AND GUIDELINES FOR THE ADA LANGUAGE

5.1 INTENT

The intent of this standard is to facilitate portability of data production software and to control costs at the DAACs.

5.2 STANDARDS

The following are the list of mandatory standards for data production software generated in the Ada language:

5.2.1 Comply with National Standard

The software shall comply with the national standard specification for Ada (MIL-STD-1815-A). The rationale for this standard is portability.

5.2.2 Use SDP Toolkit Calls

External calls from the operational data production software in the DAAC, for system and resource accesses, file I/O requests, error message transaction, and metadata formatting, shall be made through SDP Toolkit calls. The Pragma Interface Statement shall be used to interface with all commercial library software provided as part of the SDP Toolkit. Ada bindings for the SDP Toolkit will not be developed and it is the software developer's responsibility to utilize the FORTRAN or C version of the SDP Toolkit correctly. The rationale for this standard is portability as well as having one group develop this code instead of all science teams duplicating development effort.

5.2.3 Prohibited SDP Toolkit Calls

The delivered code shall not make SDP Toolkit calls from within Ada tasks. The rationale for this standard is operability and maintainability.

5.2.4 Ada Library Support

There are no Ada libraries supported. All software shall be delivered in source code form.

5.3 GUIDELINES

5.3.1 Avoid Platform-Unique Features

Since Chapter 13 of MIL-STD-1815-A deals specifically with features for adapting Ada to platform-unique features, Chapter 13 features should be used with care. Such code should be isolated as much as possible and commented extensively to make the task of porting to a new host as easy as possible.

5.3.2 Ada Style References

There are references for the production of good Ada code. The following reference might be useful in developing local coding guidelines and styles:

Seidewitz E., et al., Ada Style Guide, Version 1.1, GSFC SEL 87-002, 1987.

6. MODULE IDENTIFICATION STANDARD

6.1 INTENT

Module Identifications are needed to assist in the Science Software Integration and Test (SSI&T) process at the DAAC.

6.2 STANDARDS

To allow identification of individual items of code, a header (prolog), as defined below, shall be inserted at the top of each module in the Fortran, C or Ada production software. A module is a main program, subroutine, procedure, function, etc. This header shall also be included at the top of insert/include files, with the exception that the blocks relating to input and output parameters are omitted. Note: The example uses the C language style of comments, and would need to be converted for other languages. All entries preceded with "!" are mandatory with the exceptions: 1) of "!Team-unique Header:" if there is no team-unique header; and 2) of "!Input/Output Parameters:" if there are none of them in the module.

For heritage code, where the generation of this header information for every module is an unreasonable burden, an alternative approach based on compilation units can be used. A single header can be used to record the description and version history for all modules contained within a file which is compiled as a unit. In this case the only requirement for each module is to describe each input/output parameter (not globals). Although this alternative approach is acceptable for heritage code, the former approach, with a header for each module, is required for all new code.

The inclusion of the headers will allow automated tools within the DAAC SSI&T environment to manipulate the software items and will ease understanding by the SSI&T team.

6.3 EXAMPLE

```
line no.
00 geonav(float pos[3],float smat[3][3],float coef[6],float sun[3],
01 int nsta,int ninc,int npix,float xlat[409],float xlon[409],
02 float solz[409],float sola[409],float senz[409],float sena[409])
03/*
04!C************************************************************
05
06 !Description: This subroutine calculates the sensor
07 orientation from the orbit position vector and input values of
08 attitude offset angles. The calculations assume that the angles
09 represent the yaw, roll and pitch offsets between the local
10 vertical reference frame (at the spacecraft position) and the
11 sensor frame. The outputs are the matrix which represents
12 the transformation from the geocentric rotating to sensor
13 frame, and the coefficients which represent the Earth scan
```

14 track in the sensor frame. The reference ellipsoid uses an
15 equatorial radius of 6378.137 km and a flattening factor of
16 1/298.257 (WGS 1984).05
17
18 !Input Parameters:
19 pos[3] satellite position
20 coef[6] scan line coefficients
21 sun[3] unit sun vector in geocentric rotating coordinates
22 nsta number of first pixel to start with
23 ninc increment between pixels for computations
24 npix number of pixels in scan line
25
26 !Output Parameters:
27 xlat[409] latitude values
28 xlon[409] longitude values
29 solz[409] solar zenith values
30 sola[409] solar azimuth values
31 senz[409] sensor zenith values
32 sena[409] sensor azimuth values
33
34 !Input/Output Parameters:
35 smat[3] [3] sensor orientation matrix
36 !Revision History:
37 $LOG: geonav.c,v $
38 Revision 01.01 1993/10/12 10:12:28
39 Z. GREEN (zgreen@harp.gsfc.nasa.gov)
40 Initial delivery of software. Modified to comply with ESDIS
41 standards. It was a breeze.
42
43 Revision 01.00 1993/04/29 17:12:28
44 A. SMITH (asmith@harp.gsfc.nasa.gov)
45 Initial debugged version, based on original FORTRAN 77
46 subroutine developed in 1983 by C. ADAMS of the TELLUS
47 project.
48
49 !Team-unique Header:
50 <Science team puts any thing they want in this portion of prolog>
51 !END**********
*/
<code follows here>

The following notes describe how to use the header block, using the above example as a reference.

Line 00: Name of main procedure, include file or subroutine/ procedure/function. If this is not the main procedure or an include file, it should contain the function statement.

Line 04: Start of prolog. Initial marker can take the following values:

!FXY - contains FORTRAN XY (XY = 77 or 90) executable statements

!C - contains C executable statements

--!ADA - contains Ada executable statements

!FXY-INC - FORTRAN XY (XY = 77 or 90) include file

!C-INC - C include file

Alternately, the prolog may use a language-independent set of markers. These

include:

!PROLOG

!PROLOG-INC

Line 06: A concise but complete summary of the overall function of the module. Any references for methods and/or algorithms should be included. Use as many lines as necessary.

Line 18: Header for input. The word "parameters" in the example is a place holder; a choice of the words "parameters", "variables", or "arguments" may be used or the word may be omitted.

Line 19-24: Input parameters, arguments, or variables (not global variables) in the order they are presented to the module with a short 1-2 line description of the parameter (and its units where appropriate). Global variables should be described in the module in which they are declared (i.e., only once).

Line 26: Header for output. The word "parameters" in the example is a place holder; a choice of the words "parameters", "variables", or "arguments" may be used or the word may be omitted.

Line 27-32: Output parameters, arguments, or variables (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters.

Line 34: Header for input/output. The words "!Input/Output" are mandatory, if there are any such parameters. The word "parameters" in the example is a place holder; a choice of the words "parameters", "variables", or "arguments" may be used or the word may be omitted.

Line 35: Parameters, arguments, or variables that serve both as input parameters and as output parameters (not global variables) in the order they are contained in the function statement. Global variables should be described in the module in which they are declared (i.e., only once). Same format as for input parameters. The automated I&T tools will allow this part of the header to be omitted.

Line 36: Start of Revision History Log.

Line 37: If you are using an automated tool for revision control, you should insert any statements required immediately after the Modification History Log Header.

Line 38-47: Each revision shall contain as a minimum the revision number, date, and a short description of ALL the changes made. For larger teams, the person's name and their email address are recommended. The first entry shall include the original author and date of the code. Revisions shall be ordered with the latest first. [Note: this revision information can be supplemented with more detailed comments in the code referencing the revision number.]

Any revision numbering system relevant to your site and configuration control mechanism may be used but the "nn.mm" format is recommended, where "mm" is updated each time a change is made to the module and "nn" is updated when the function of the module changes or the algorithm/method is changed. [Note: The release number for the data production software is not related to the revision numbers on individual modules - the release number scheme should be determined by the development team. The date associated with a release is more meaningful than a release number to those outside the development team.]

Line 49: Start of Team-unique Header.

Line 50: Each team may design it's own header section(s). There may be more than one of these, mixed in with the mandatory header sections of the prologue. The automated I&T tools will ignore this part of the header.

Line 51: End of source code prolog.

General Notes:

1. The marker may have the format "!marker" or "!marker". The marker is case insensitive (e.g., Marker, marker, and MARKER are the same).

2. A team-unique marker does not have to have the phrase "team unique" in the marker.

3. An Ada prolog starts with "--!" rather than "!".

7. SCRIPT LANGUAGE STANDARD

7.1 INTENT

The intent of this standard is to facilitate software portability and to control costs at the DAAC.

7.2 STANDARDS

7.2.1 Shell Languages

Command language code delivered to the DAACs as part of the science data production software shall be written using one or more of the following script languages.

a.) csh,

b.) ksh,

c.) perl,

d.) POSIX-compatible shell language,

e.) bourne shell.

Adherence to these standards is mandatory for script command languages that are used to generate the command language portion of the science data production software delivered to the DAACs.

The POSIX standard is published in IEEE Std 1003.2-1992.

7.2.2 Shell Names

The following case-independent filename extensions shall be used for all science software script files delivered to the DAAC as part of the science software delivery:

a.) c script .csh

b.) korn script .ksh

c.) perl script .pl or .perl

d.) bourne script .sh

7.3 GUIDELINES

7.3.1 Minimize Number of Script Languages

Command language code delivered to the DAACs as part of the science data production software from a science team should be written using the smallest possible number of script languages.

423-16-01

7.3.2 Use Efficient Mixture of Compiled and Interpreted Programming Languages

The use of compiled programming language should be maximized and the use of interpreted script command language should be minimized, in developing science data production software for delivery to the DAACs. The rationale for this guideline is efficiency of software execution and adherence to standards for portability. Compiled code executes more efficiently than interpreted code. There are formal national standards for the approved compiled programming languages while there is only one formal national standard, POSIX, for a script language.

APPENDIX A. ESDIS POLICY REGARDING PROHIBITED FUNCTIONS IN SCIENCE DATA PRODUCTION SOFTWARE

A.1 GENERAL PRINCIPLES

The general guiding principles behind the prohibited functions will apply to both scripts and compiled code.

A.1.1 Avoid Interactivity

Since a running Product Generation Executive (PGE) is not connected to any terminal device (screen, keyboard, mouse, etc.), the PGE should not invoke any function, routine, or utility that requires such a device. The PGEs: 1) may not write to standard output (stdout) or standard error (stderr), or read from standard input (stdin); 2) may not attempt to display anything to the screen (GUIs, prompts, messages); and 3) may not attempt to read anything from an input device such as a keyboard or mouse. This necessarily means that a PGE cannot be directly interactive with a user or operator.

Examples: 1) do not use I/O streams stdin, stdout, or stderr in C; 2) do not use file handles STDIN, STDOUT, or STDERR in Perl; 3) do not write to unit 6 or *, or read from unit 5 or *, in FORTRAN 77 or Fortran 90 and 4) do not use unredirected or unpiped stdin, stdout, or stderr in shell scripts.

A.1.2 Use Process Control Files

All I/O in a PGE must be through the use of files which are defined in the Process Control File (PCF). This applies to both compiled code and to scripts. The SDP Toolkit provides tools for both.

A.1.3 No Direct Control of Files

PGEs must not attempt to circumvent the SDP Toolkit in order to manage files or file systems directly.

Examples: Do not use chdir(), mkdir(), mv, cd, rm, rmdir, chmod, chown

A.1.4 No Direct Control of Processes

PGEs must not attempt to circumvent the SDP Toolkit in order to manage or control processes.

Examples: Do not use kill, su, wait(), nice, sleep, umask()

A.1.5 No Direct Control of Network

PGEs must not attempt to initiate network connections or transfers directly.

Examples: Do not use ftp, telnet, rlogin, lp, socket(), RCP

A.1.6 No Direct Control of Shell

PGEs must not interfere in or otherwise thwart the shell environment in which the PGEs are running. Environment variables used by the Toolkit (e.g. PGSHOME, PGSBIN, PGSMSG, PGSLIB, HDFHOME, HDFINC, etc.) should not be redefined. The environment variables that are typical in a user shell (e.g. USER, PATH, SHELL, path, HOSTNAME, MACHINETYPE, SHLVL, PWD, DISPLAY, TERM, etc.) should not be redefined.

A.2 LANGUAGE-SPECIFIC LISTS

For each language, the prohibited functions include but are not limited to the items in the following lists. It is impossible to list all possible functions, routines, utilities, etc. that someday may present a problem to the ECS.

A.2.1 C Language Prohibited Functions

abort()
access()
alarm()
Any use of file stream stdin
Any use of file stream stdout
Any use of file stream stderr
atexit()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
clearerr()
close()
closedir()
creat()
ctermid()
dup()
dup2()
exec()
_exit()
fclose()
fcntl()
fdopen()
fileno()

fopen()
fork()
fpathconf()
freopen()
fstat()
getchar()
getcwd()
getegid()
geteuid()
getgid()
getgrgid()
getgrnam()
getgroups()
getlogin()
getpgrp()
getpid()
getppid()
getpwnam()
getpwuid()
gets()
getuid()
isatty()
kill()
localeconv()
link()
lseek()
mkdir()
mkfifo()
opendir()
open()
pathconf()
pause()
pipe()
printf()
putchar()
puts()
read()
readdir()
remove()
rename()
rewind()
rewinddir()

rmdir()
scanf()
setbuf()
setgid()
setgrgid()
setlocale()
setpgid()
setsid()
setuid()
sig...()
sleep()
stat()
system()
tc...()
tmpfile()
tmpnam()
ttyname()
uname()
unlink()
umask()
utime()
vprintf()
uname()
wait()
waitpid()
write()

A.2.2 FORTRAN (77 & 90) Language Prohibited Functions.

Any use of output to unit 6 or *
Any use of input from unit 5 or *
CLOSE()
OPEN()
PRINT * ...
PXFACCESS()
PXFALARM()
PXFCFGETISPEED()
PXFCFGETOSPEED()
PXFCFSETISPEED()
PXFCFSETOSPEED()
PXFCHMOD()
PXFCHOWN()

PXFCLOSE()
PXFCLOSEDIR()
PXFCNTL()
PXFCREAT()
PXFCTERMID()
PXFDUP()
PXFDUP2()
PXFCHDIR()
PXFEXEC...()
PXFEXIT()
PXFFASTEXIT()
PXFFCNTL()
PXFFDOPEN()
PXFFILENO()
PXFFORK()
PXFFPATHCONF()
PXFFSTAT()
PXFGETCWD()
PXFGETEGID()
PXFGETEUID()
PXFGETGID()
PXFGETGRGID()
PXFGETGRNAM()
PXFGETGROUPS()
PXFGETLOGIN()
PXFGETPGRP()
PXFGETPID()
PXFGETPPID()
PXFGETPWNAM()
PXFGETPWUID()
PXFGETUID()
PXFIS...()
PXFISATTY()
PXFKILL()
PXFLINK()
PXFLSEEK()
PXFMKDIR()
PXFMKFIFO()
PXFOPEN()
PXFOPENDIR()
PXFPAUSE()
PXFPATHCONF()

PXFPIPE()
PXFPOSIXIO()
PXFREAD()
PXFREADDIR()
PXFRENAME()
PXFRMDIR()
PXFREWINDDIR()
PXFSETGID()
PXFSETPGID()
PXFSETSID()
PXFSETUID()
PXFSIG...()
PXFSLEEP()
PXFSTAT()
PXFTC...()
PXFTTYNAME()
PXFUMASK()
PXFUNAME()
PXFUNLINK()
PXFUTIME()
PXFWAIT()
PXFWAITPID()
PXFWRITE()
READ * ...
READ(*,...)
READ(5,...)
WRITE(*,...)
WRITE(6,...)

A.2.3 Bourne, C, Korn, and Perl Shell Prohibited Functions and Utilities.

Any use of standard input (stdin)
Any use of standard output (stdout)
Any use of standard error (stderr)
at, atq, atrm
cd
chgrp
chmod
chown
cp
find
ftp

kill
ln
lp, lpr, lpstat
mail (and any related functions)
mkdir
nice
printf
rcp
read
rlogin
rm
rmdir
rsh
script
sleep
su
telnet
touch
umask
write

A.2.4 Perl-Specific Prohibited Functions.

accept()
alarm()
Any use of file handle STDIN
Any use of file handle STDOUT
Any use of file handle STDERR
bind()
chdir()
closedir()
connect()
chmod()
chown()
chroot()
dbmclose()
dbmopen()
die
dump()
exec()
fcntl()
fileno()

flock()
fork()
getgrgid()
getlogin()
getpeername()
getpgrp()
getppid()
getpriority()
getpwuid()
getsockname()
getsockopt()
link()
listen()
kill()
mkdir()
msgctl()
msgget()
msgsnd()
opendir()
readdir()
readlink()
recv()
rename()
rewinddir()
rmdir()
seekdir()
setpgrp()
setpriority()
setsockopt()
semctl()
semget()
semop()
send()
shmctl()
shmget()
shmread()
shmwrite()
shutdown()
sleep()
socket()
socketpair()
symlink()

syscall()
system()
telldir()
unlink()
umask()
utime()
wait()
waitpid()
warn()

APPENDIX B. ESDIS POLICY FOR SOFTWARE STANDARDS CHANGES AND WAIVERS

B.1 SCOPE

This discussion is limited to the ESDIS Software Standards and Guidelines for Science Team SCFs and for Science team software targeted for the DAAC. These standards are documented in this document. This policy does not apply to any documents that describe good programming practices (corollary information) or other recommended practices. However, guidelines (including standards) in this document are controlled by the ESDIS Project Configuration Control Board (CCB).

There are two policy categories when considering changes or exceptions to the ESDIS Standards. These categories are: permanent changes to the Standards, and individual exceptions to the Standards via waivers. These are discussed separately.

B.1.1 Standards Changes

These may be suggested, not requested, (i.e. no formal response to the originating party is required) by any involved or associated party. The Standards Committee (science team, DAAC, ECS contractor, and ESDIS personnel) will: 1) hold a caucus to discuss the suggestion; 2) write up the changes to the Standards Document; 3) circulate these changes within the ESDIS Standards Committee, for concurrence; and then 4) transfer the revised document recommendation to the formal ESDIS CCB for final approval.

All CCB approved changes to the Standards Document will then be broadcast to EOS Instrument, Interdisciplinary, and DAAC teams. Changes to the Standards Document will be made by the ESDIS Project Configuration Management Officer using change bars and revision markings within the body of the Standards Document. Reasons for Standards changes will be disseminated to the EOS teams, but will not be included in the Standards Document.

The rationale for Standards and/or Guidelines changes can include: code maintenance, portability, reusability, ease of creation, heritage considerations, and common coding practices.

The following definitions have been used for the above terms:

a.) Maintenance: long term viability of the code (20 years). This applies to code headers, documentation, understandability, modularity, etc.

b.) Portability: ease of moving code to new platforms. This applies to the compatibility of languages and coding techniques across all computer architectures.

c.) Reusability: the ability of others to use and understand the code as written. This applies more to the calling sequence, parameter passing, and utility and library aspects of the code. This term also considers the quality of the code documentation.

d.) Ease of creation: industry accepted coding practices including macros, make files, handling of machine dependencies, debugging, profiling, and other integrated development environment (IDE) aspects.

e.) Heritage considerations: allowing old coding practices in previously written code that has been thoroughly debugged, exercised, in wide usage, and would create redundant work to re-invent.

f.) Common coding practices: techniques that the programming community as a whole utilizes and are therefore assumed to be understandable by all programmers.

Note that heritage code will be brought up to standards before delivery to the DAAC. Exceptions to this policy will be covered as temporary waivers to the Standards.

B.1.2 Individual Waivers to the Standard

Waivers are defined to be temporary exceptions to the Standards Document and are an agreement between ESDIS and an individual. Depending upon the nature of the waiver, it may apply to the entire EOSDIS community. All waivers will have an expiration date. Criteria for granting or renewing a waiver include feasibility in the ECS environment and cost impact.

The procedure for waivers is for the request to be submitted to an ESDIS Science Software Manager (SSM). The SSM will acknowledge the request and give an expected time frame for an official response to the waiver request. The ESDIS Standards Committee will convene to create an unofficial acknowledgment to the requester indicating that the request is reasonable or not, pro's and con's discussing the merits of the request, and a proposed date at which the response will be officially approved. This is necessary to give a timely response to the requester, who can then proceed with accommodating the proposed waiver results.

All waivers will need approval of the ESDIS Standards Committee and, if appropriate, of the ESDIS Data Processing Manager before becoming official, but will not need ESDIS CCB approval.

A waiver rejection appeal process can be initiated by the involvement of a third party, at the discretion of the ESDIS Data Processing Manager, to further champion the waiver. Third parties may be from any of the ESDIS associated entities, such as a DAAC, ECS contractor, instrument team, ECS component (SDPS, CSMS, etc.), spacecraft platform developer, or EOSDIS working group. Waiver appeal approval will be determined by a consensus among the instrument teams and DAAC representatives, heavily weighted with the opinion of the ESDIS Data Processing Manager.

Criteria for waivers will emphasize the short duration for which a waiver is to be granted. The granting or renewal of a waiver would cover only specified deliveries with the requirement that the code would be brought into standards compliance by a specified date or milestone.

Waivers are renewable as long as the instrument team continues to need the waiver and the DAAC can continue to support the exception to the standards.

A list of the current waivers may be obtained from an ESDIS SSM.

ACRONYMS AND ABBREVIATIONS

API	Application Program Interface
CCB	Configuration Control Board
DAAC	Distributed Active Archive Center
DPFT	Data Processing Focus Team
ECS	EOSDIS Core System
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
ESDIS	Earth Science Data and Information System
IDE	Integrated Development Environment
I/O	Input/Output
PCF	Process Control File
PGE	Product Generation Executive
SCF	Science Computing Facility
SDP	Science Data Production
SSI&T	Science Software Integration and Test
SSM	Science Software Manager